10.4 (2,4) Trees

Some data structures we discuss in this chapter, including $(2,4)$ trees, are multi-way search trees, that is, trees with internal nodes that have two or more children. Thus, before we define $(2,4)$ trees, let us discuss multi-way search trees.

10.4.1 Multi-Way Search Trees

Recall that multi-way trees are defined so that each internal node can have many children. In this section, we discuss how multi-way trees can be used as search trees. Recall that the ***entries*** that we store in a search tree are pairs of the form (k,x), where k is the ***key*** and x is the value associated with the key. However, we do not discuss how to perform updates in multi-way search trees now, since the details for update methods depend on additional properties we want to maintain for multi-way trees, which we discuss in Section 14.3.1.

Definition of a Multi-way Search Tree

Let v be a node of an ordered tree. We say that v is a ***d-node*** if v has d children. We define a ***multi-way search tree*** to be an ordered tree T that has the following properties, which are illustrated in Figure 10.1(a):

- Each internal node of T has at least two children. That is, each internal node is a d-node such that $d \geq 2$.
- Each internal d-node v of T with children $v_1,\ldots,v_d$ stores an ordered set of $d-1$ key-value entries $(k_1,x_1),\ldots,(k_{d-1},x_{d-1})$, where $k_1 \leq \cdots \leq k_{d-1}$.
- Let us conventionally define $k_0 = -\infty$ and $k_d = +\infty$. For each entry (k,x) stored at a node in the subtree of v rooted at v_i, $i = 1,\ldots,d$, we have that $k_{i-1} \leq k \leq k_i$.

That is, if we think of the set of keys stored at v as including the special fictitious keys $k_0 = -\infty$ and $k_d = +\infty$, then a key k stored in the subtree of T rooted at a child node v_i must be "in between" two keys stored at v. This simple viewpoint gives rise to the rule that a d-node stores $d-1$ regular keys, and it also forms the basis of the algorithm for searching in a multi-way search tree.

By the above definition, the external nodes of a multi-way search do not store any entries and serve only as "placeholders," as has been our convention with binary search trees (Section 10.1); hence, a binary search tree can be viewed as a special case of a multi-way search tree, where each internal node stores one entry and has two children. In addition, while the external nodes could be *null*, we make the simplifying assumption here that they are actual nodes that don't store anything.

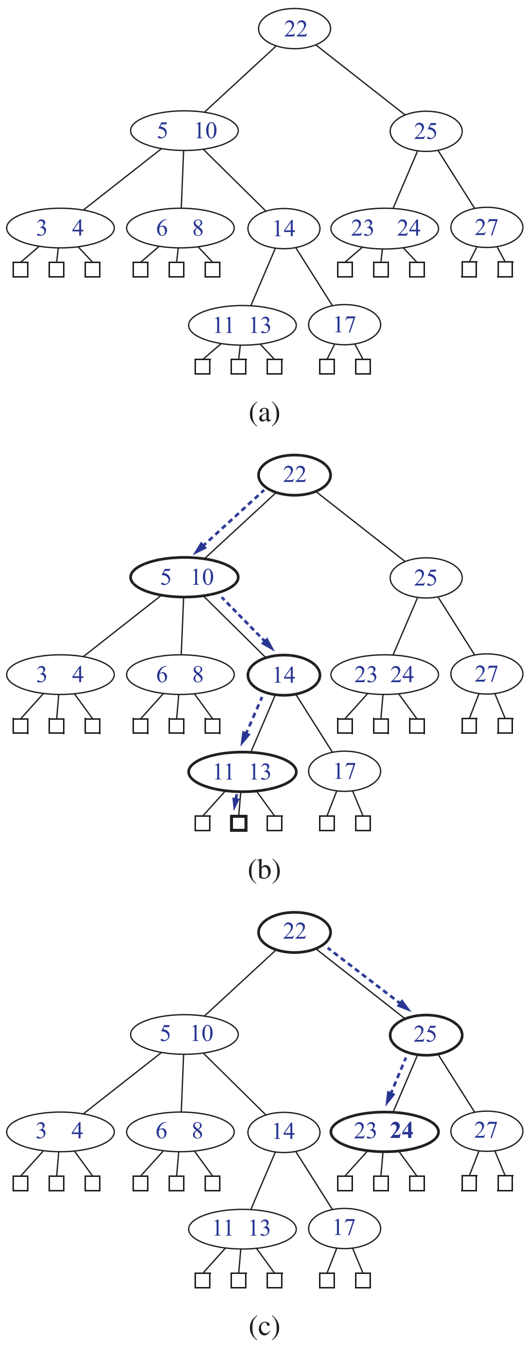


Figure 10.20: (a) A multi-way search tree T; (b) search path in T for key 12 (unsuccessful search); (c) search path in T for key 24 (successful search).

Whether internal nodes of a multi-way tree have two children or many, however, there is an interesting relationship between the number of entries and the number of external nodes.

Proposition 10.7: *An n-entry multi-way search tree has $n+1$ external nodes.*

We leave the justification of this proposition as an exercise (Exercise C-10.17).

Searching in a Multi-Way Tree

Given a multi-way search tree T, we note that searching for an entry with key k is simple. We perform such a search by tracing a path in T starting at the root. (See Figure 10.1(b) and (c).) When we are at a d-node v during this search, we compare the key k with the keys $k_1,\ldots,k_{d-1}$ stored at v. If $k = k_i$ for some i, the search is successfully completed. Otherwise, we continue the search in the child v_i of v such that $k_{i-1} < k < k_i$. (Recall that we conventionally define $k_0 = -\infty$ and $k_d = +\infty$.) If we reach an external node, then we know that there is no entry with key k in T, and the search terminates unsuccessfully.

Data Structures for Representing Multi-way Search Trees

In Section 7.1.4, we discuss a linked data structure for representing a general tree. This representation can also be used for a multi-way search tree. In fact, in using a general tree to implement a multi-way search tree, the only additional information that we need to store at each node is the set of entries (including keys) associated with that node. That is, we need to store with v a reference to some collection that stores the entries for v.

Recall that when we use a binary search tree to represent an ordered map M, we simply store a reference to a single entry at each internal node. In using a multi-way search tree T to represent M, we must store a reference to the ordered set of entries associated with v at each internal node v of T. This reasoning may at first seem like a circular argument, since we need a representation of an ordered map to represent an ordered map. We can avoid any circular arguments, however, by using the ***bootstrapping*** technique, where we use a previous (less advanced) solution to a problem to create a new (more advanced) solution. In this case, bootstrapping consists of representing the ordered set associated with each internal node using a map data structure that we have previously constructed (for example, a search table based on a sorted array, as shown in Section 9.3.1). In particular, assuming we already have a way of implementing ordered maps, we can realize a multi-way search tree by taking a tree T and storing such a map at each node of T.

The map we store at each node v is known as a ***secondary*** data structure, because we are using it to support the bigger, ***primary*** data structure. We denote the map stored at a node v of T as $M(v)$. The entries we store in $M(v)$ allow us to find which child node to move to next during a search operation. Specifically, for each node v of T, with children $v_1, \ldots, v_d$ and entries (k_1, x_1), ..., (k_{d-1}, x_{d-1}), we store, in the map $M(v)$, the entries

$$(k_1, (x_1, v_1)), (k_2, (x_2, v_2)), \ldots, (k_{d-1}, (x_{d-1}, v_{d-1})), (+\infty, (\emptyset, v_d)).$$

That is, an entry $(k_i, (x_i, v_i))$ of map $M(v)$ has key k_i and value (x_i, v_i). Note that the last entry stores the special key $+\infty$.

With the realization of the multi-way search tree T above, processing a d-node v while searching for an entry of T with key k can be done by performing a search operation to find the entry $(k_i, (x_i, v_i))$ in $M(v)$ with smallest key greater than or equal to k. We distinguish two cases:

- If $k < k_i$, then we continue the search by processing child v_i. (Note that if the special key $k_d = +\infty$ is returned, then k is greater than all the keys stored at node v, and we continue the search processing child v_d.)
- Otherwise $(k = k_i)$, then the search terminates successfully.

Consider the space requirement for the above realization of a multi-way search tree T storing n entries. By Proposition 10.7, using any of the common realizations of an ordered map (Chapter 9) for the secondary structures of the nodes of T, the overall space requirement for T is $O(n)$.

Consider next the time spent answering a search in T. The time spent at a d-node v of T during a search depends on how we realize the secondary data structure $M(v)$. If $M(v)$ is realized with a sorted array (that is, an ordered search table), then we can process v in $O(\log d)$ time. If $M(v)$ is realized using an unsorted list instead, then processing v takes $O(d)$ time. Let $d_{\max}$ denote the maximum number of children of any node of T, and let h denote the height of T. The search time in a multi-way search tree is either $O(hd_{\max})$ or $O(h \log d_{\max})$, depending on the specific implementation of the secondary structures at the nodes of T (the map $M(v)$). If $d_{\max}$ is a constant, the running time for performing a search is $O(h)$, irrespective of the implementation of the secondary structures.

Thus, the primary efficiency goal for a multi-way search tree is to keep the height as small as possible, that is, we want h to be a logarithmic function of n, the total number of entries stored in the map. A search tree with logarithmic height such as this is called a ***balanced search tree***.

Definition of a $(2,4)$ Tree

A multi-way search tree that keeps the secondary data structures stored at each node small and also keeps the primary multi-way tree balanced is the $(\mathbf{2},\mathbf{4})$ ***tree***, which is sometimes called 2-4 tree or 2-3-4 tree. This data structure achieves these goals by maintaining two simple properties (see Figure 10.21):

***Size Property*: ** Every internal node has at most four children

***Depth Property*: ** All the external nodes have the same depth

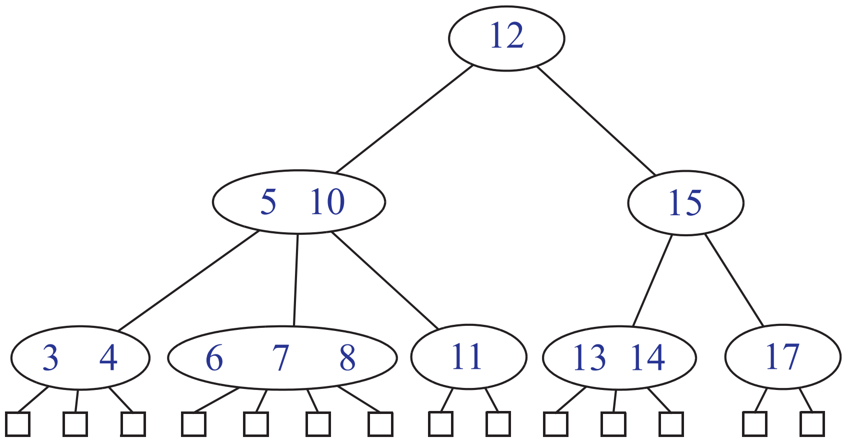


Figure 10.21: A $(2,4)$ tree.

Again, we assume that external nodes are empty and, for the sake of simplicity, we describe our search and update methods assuming that external nodes are real nodes, although this latter requirement is not strictly needed.

Enforcing the size property for $(2,4)$ trees keeps the nodes in the multi-way search tree simple. It also gives rise to the alternative name “2-3-4 tree,” since it implies that each internal node in the tree has 2, 3, or 4 children. Another implication of this rule is that we can represent the map $M(v)$ stored at each internal node v using an unordered list or an ordered array, and still achieve $O(1)$-time performance for all operations (since $d_{\max} = 4$). The depth property, on the other hand, enforces an important bound on the height of a $(2,4)$ tree.

Proposition 10.8: *The height of a $(2,4)$ tree storing n entries is $O(\log n)$.*

Justification: Let h be the height of a $(2,4)$ tree T storing n entries. We justify the proposition by showing that the claims

$$\frac{1}{2}\log(n+1) \le h \tag{10.9}$$

and

$$h \le \log(n+1) \tag{10.10}$$

are true.

To justify these claims note first that, by the size property, we can have at most 4 nodes at depth 1, at most 4^2 nodes at depth 2, and so on. Thus, the number of external nodes in T is at most 4^h. Likewise, by the depth property and the definition of a $(2,4)$ tree, we must have at least 2 nodes at depth 1, at least 2^2 nodes at depth 2, and so on. Thus, the number of external nodes in T is at least 2^h. In addition, by Proposition 10.7, the number of external nodes in T is $n+1$. Therefore, we obtain

$$2^h \le n+1$$

and

$$n+1 \le 4^h.$$

Taking the logarithm in base 2 of each of the above terms, we get that

$$h \le \log(n+1)$$

and

$$\log(n+1) \le 2h,$$

which justifies our claims (10.9 and 10.10). ■

Proposition 10.8 states that the size and depth properties are sufficient for keeping a multi-way tree balanced (Section 10.4.1). Moreover, this proposition implies that performing a search in a $(2,4)$ tree takes $O(\log n)$ time and that the specific realization of the secondary structures at the nodes is not a crucial design choice, since the maximum number of children $d_{\max}$ is a constant (4). We can, for example, use a simple ordered map implementation, such as an array-list search table, for each secondary structure.

10.4.2 Update Operations for $(2,4)$ Trees

Maintaining the size and depth properties requires some effort after performing insertions and removals in a $(2,4)$ tree, however. We discuss these operations next.

Insertion

To insert a new entry (k,x), with key k, into a $(2,4)$ tree T, we first perform a search for k. Assuming that T has no entry with key k, this search terminates unsuccessfully at an external node z. Let v be the parent of z. We insert the new entry into node v and add a new child w (an external node) to v on the left of z. That is, we add entry (k,x,w) to the map $M(v)$.

Our insertion method preserves the depth property, since we add a new external node at the same level as existing external nodes. Nevertheless, it may violate the size property. Indeed, if a node v was previously a 4-node, then it may become a 5-node after the insertion, which causes the tree T to no longer be a $(2,4)$ tree. This type of violation of the size property is called an ***overflow*** at node v, and it must be resolved in order to restore the properties of a $(2,4)$ tree. Let $v_1,\ldots,v_5$ be the children of v, and let $k_1,\ldots,k_4$ be the keys stored at v. To remedy the overflow at node v, we perform a ***split*** operation on v as follows (see Figure 10.22):

- Replace v with two nodes v' and v'', where
  - ◦ v' is a 3-node with children v_1,v_2,v_3 storing keys k_1 and k_2
  - ◦ v'' is a 2-node with children v_4,v_5 storing key k_4
- If v was the root of T, create a new root node u; else, let u be the parent of v
- Insert key k_3 into u and make v' and v'' children of u, so that if v was child i of u, then v' and v'' become children i and $i+1$ of u, respectively

We show a sequence of insertions in a $(2,4)$ tree in Figure 10.23.

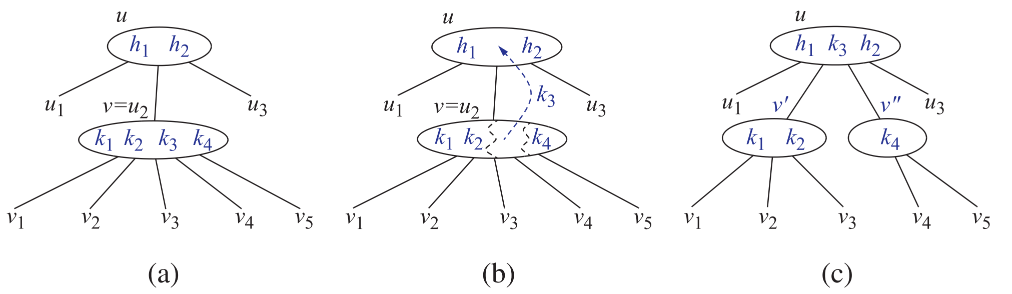


Figure 10.22: A node split: (a) overflow at a 5-node v; (b) the third key of v inserted into the parent u of v; (c) node v replaced with a 3-node v' and a 2-node v''.

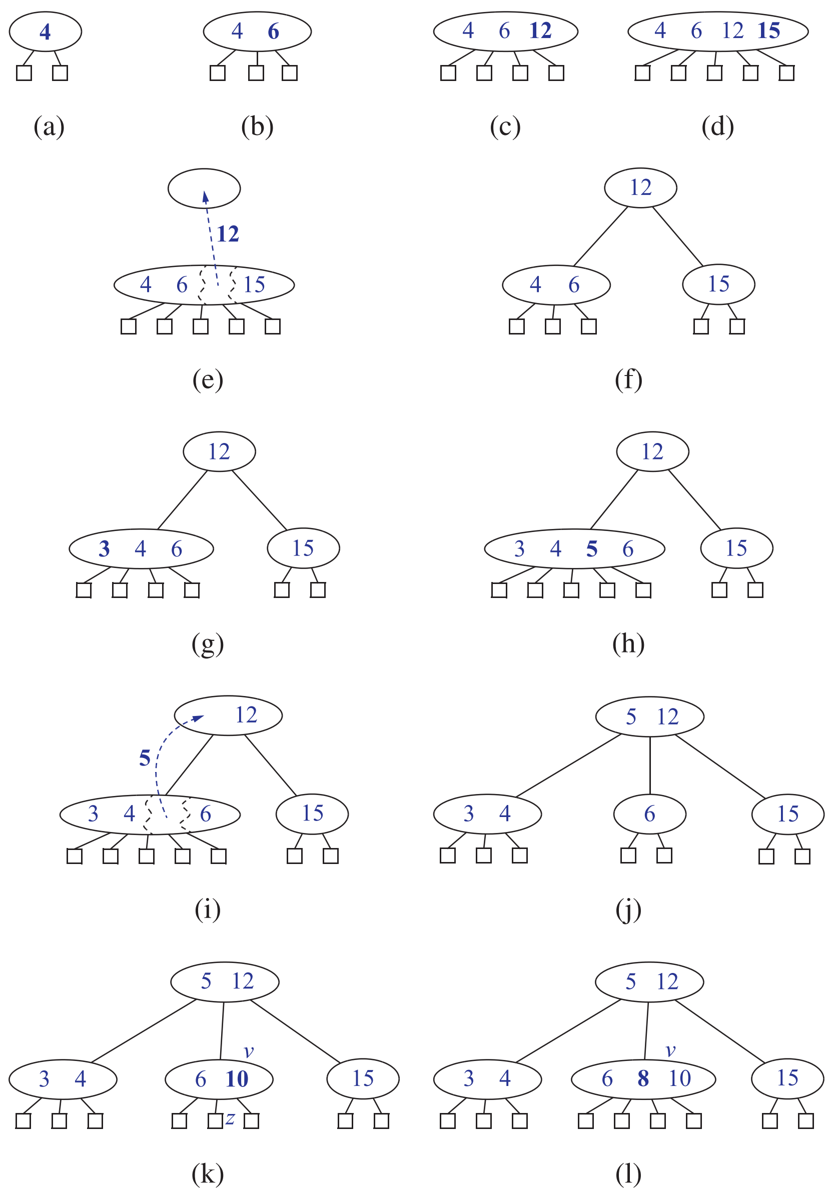


Figure 10.23: A sequence of insertions into a (2,4) tree: (a) initial tree with one entry; (b) insertion of 6; (c) insertion of 12; (d) insertion of 15, which causes an overflow; (e) split, which causes the creation of a new root node; (f) after the split; (g) insertion of 3; (h) insertion of 5, which causes an overflow; (i) split; (j) after the split; (k) insertion of 10; (l) insertion of 8.

Analysis of Insertion in a $(2,4)$ Tree

A split operation affects a constant number of nodes of the tree and $O(1)$ entries stored at such nodes. Thus, it can be implemented to run in $O(1)$ time.

As a consequence of a split operation on node v, a new overflow may occur at the parent u of v. If such an overflow occurs, it triggers a split at node u in turn. (See Figure 10.24.) A split operation either eliminates the overflow or propagates it into the parent of the current node. Hence, the number of split operations is bounded by the height of the tree, which is $O(\log n)$ by Proposition 10.8. Therefore, the total time to perform an insertion in a $(2,4)$ tree is $O(\log n)$.

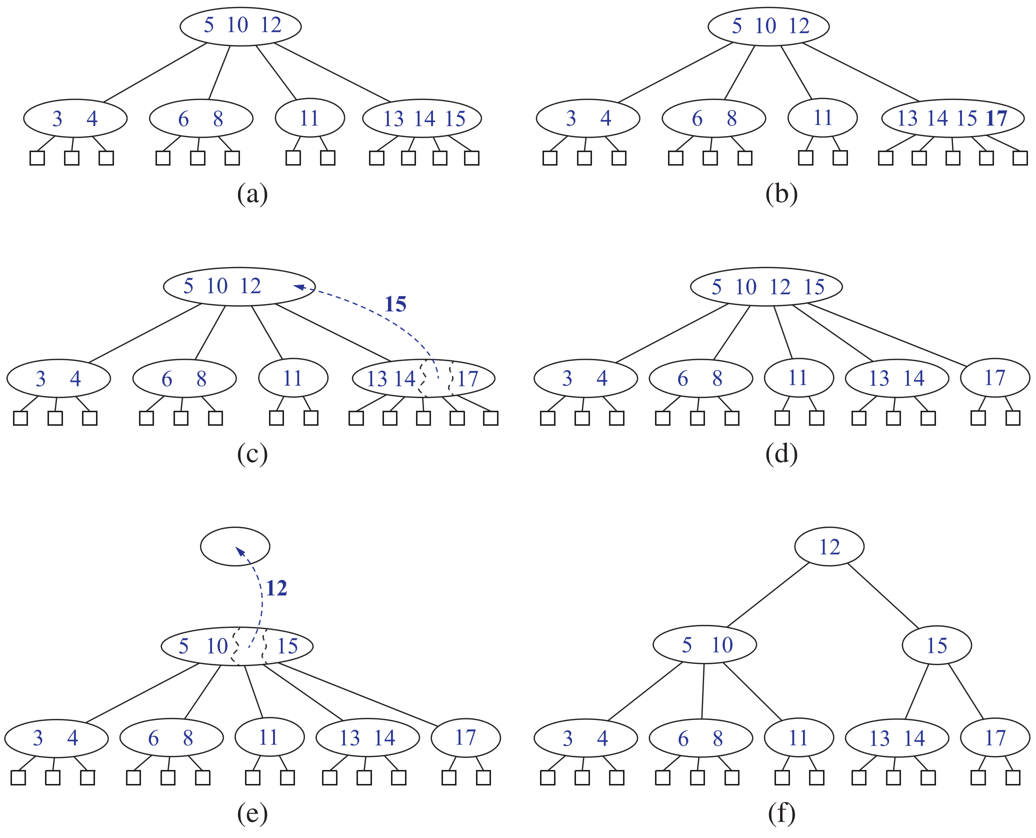


Figure 10.24: An insertion in a $(2,4)$ tree that causes a cascading split: (a) before the insertion; (b) insertion of 17, causing an overflow; (c) a split; (d) after the split a new overflow occurs; (e) another split, creating a new root node; (f) final tree.

Removal

Let us now consider the removal of an entry with key k from a $(2,4)$ tree T. We begin such an operation by performing a search in T for an entry with key k. Removing such an entry from a $(2,4)$ tree can always be reduced to the case where the entry to be removed is stored at a node v whose children are external nodes. Suppose, for instance, that the entry with key k that we wish to remove is stored in the ith entry (k_i, x_i) at a node z that has only internal-node children. In this case, we swap the entry (k_i, x_i) with an appropriate entry that is stored at a node v with external-node children as follows (see Figure 10.25(d)):

1. We find the right-most internal node v in the subtree rooted at the ith child of z, noting that the children of node v are all external nodes.
2. We swap the entry (k_i, x_i) at z with the last entry of v.

Once we ensure that the entry to remove is stored at a node v with only external-node children (because either it was already at v or we swapped it into v), we simply remove the entry from v (that is, from the map $M(v)$) and remove the ith external node of v.

Removing an entry (and a child) from a node v as described above preserves the depth property, because we always remove an external node child from a node v with only external-node children. However, in removing such an external node we may violate the size property at v. Indeed, if v was previously a 2-node, then it becomes a 1-node with no entries after the removal (Figure 10.25(d) and (e)), which is not allowed in a $(2,4)$ tree. This type of violation of the size property is called an ***underflow*** at node v. To remedy an underflow, we check whether an immediate sibling of v is a 3-node or a 4-node. If we find such a sibling w, then we perform a ***transfer*** operation, in which we move a child of w to v, a key of w to the parent u of v and w, and a key of u to v. (See Figure 10.25(b) and (c).) If v has only one sibling, or if both immediate siblings of v are 2-nodes, then we perform a ***fusion*** operation, in which we merge v with a sibling, creating a new node v', and move a key from the parent u of v to v'. (See Figure 10.26(e) and (f).)

A fusion operation at node v may cause a new underflow to occur at the parent u of v, which in turn triggers a transfer or fusion at u. (See Figure 10.26.) Hence, the number of fusion operations is bounded by the height of the tree, which is $O(\log n)$ by Proposition 10.8. If an underflow propagates all the way up to the root, then the root is simply deleted. (See Figure 10.26(c) and (d).) We show a sequence of removals from a $(2,4)$ tree in Figures 10.25 and 10.26.

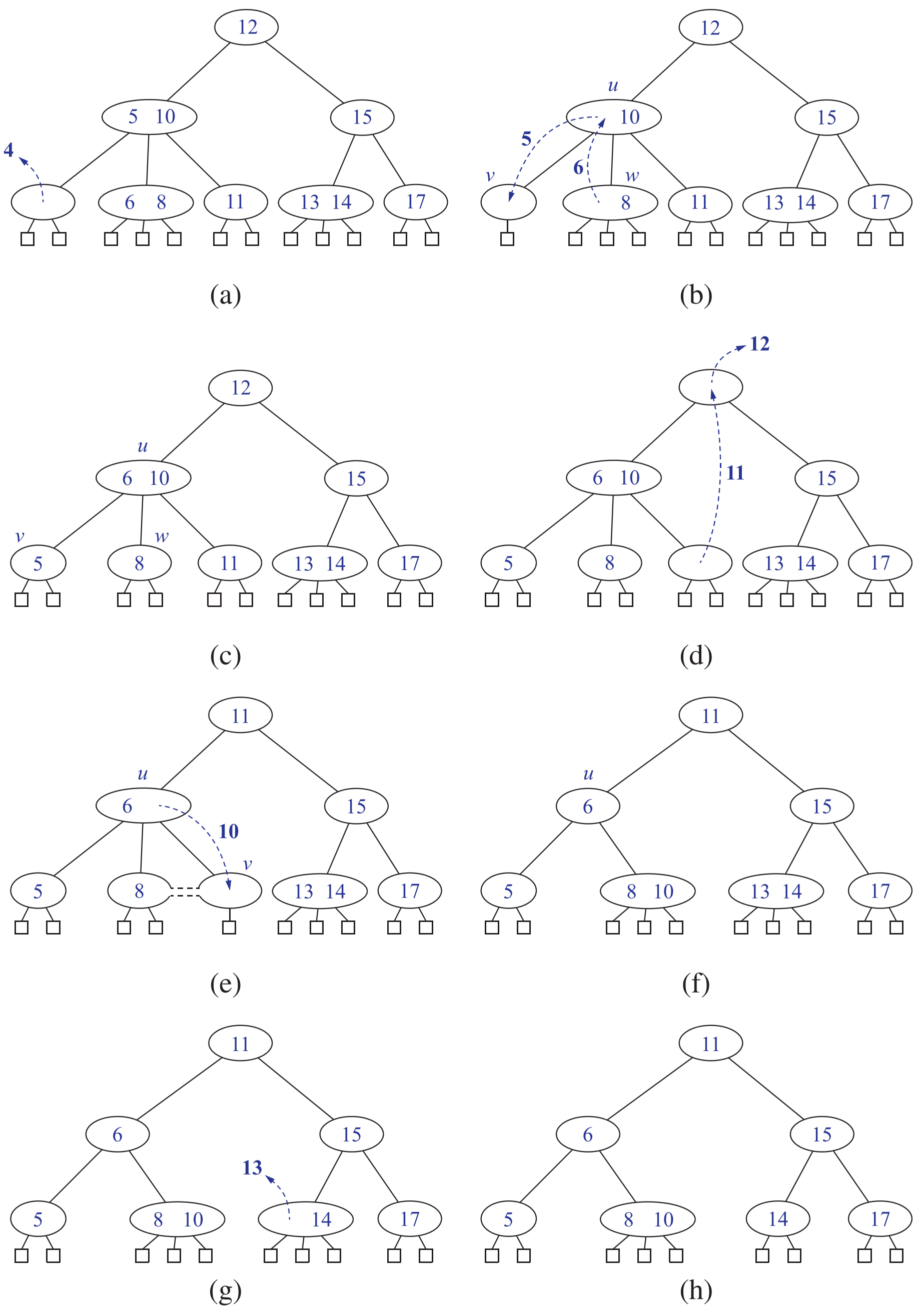


Figure 10.25: A sequence of removals from a $(2,4)$ tree: (a) removal of 4, causing an underflow; (b) a transfer operation; (c) after the transfer operation; (d) removal of 12, causing an underflow; (e) a fusion operation; (f) after the fusion operation; (g) removal of 13; (h) after removing 13.

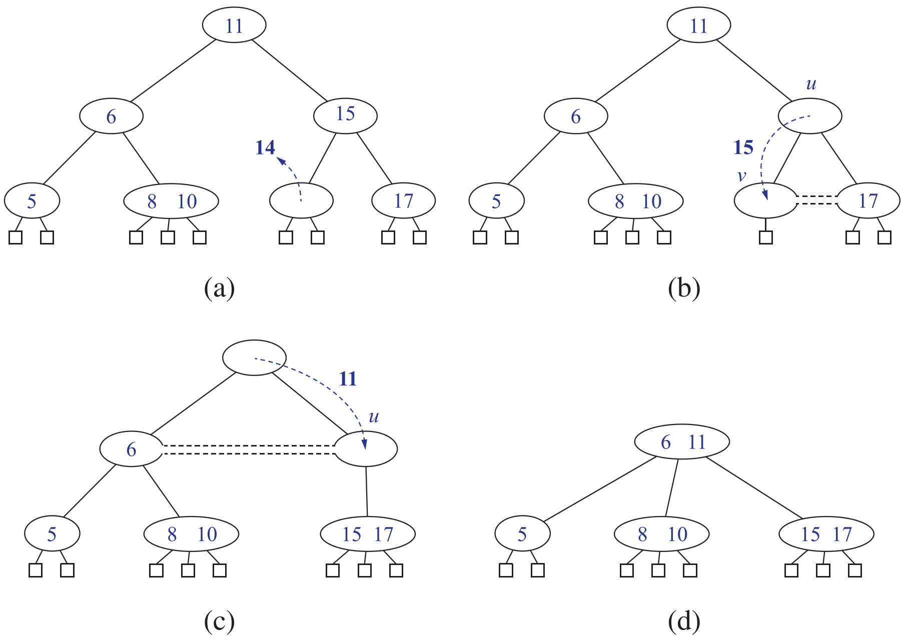


Figure 10.26: A propagating sequence of fusions in a $(2,4)$ tree: (a) removal of 14, which causes an underflow; (b) fusion, which causes another underflow; (c) second fusion operation, which causes the root to be removed; (d) final tree.

Performance of $(2,4)$ Trees

Table 10.3 summarizes the running times of the main operations of a map realized with a $(2,4)$ tree. The time complexity analysis is based on the following:

- The height of a $(2,4)$ tree storing n entries is $O(\log n)$, by Proposition 10.8
- A split, transfer, or fusion operation takes $O(1)$ time
- A search, insertion, or removal of an entry visits $O(\log n)$ nodes.

Operation	***Time***
size, empty	$O(1)$
find, insert, erase	$O(\log n)$

Table 10.3: Performance of an n-entry map realized by a $(2,4)$ tree. The space usage is $O(n)$.

Thus, $(2,4)$ trees provide for fast map search and update operations. $(2,4)$ trees also have an interesting relationship to the data structure we discuss next.

10.5 Red-Black Trees

Although AVL trees and $(2,4)$ trees have a number of nice properties, there are some map applications for which they are not well suited. For instance, AVL trees may require many restructure operations (rotations) to be performed after a removal, and $(2,4)$ trees may require many fusing or split operations to be performed after either an insertion or removal. The data structure we discuss in this section, the red-black tree, does not have these drawbacks, however, as it requires that only $O(1)$ structural changes be made after an update in order to stay balanced.

A ***red-black tree*** is a binary search tree (see Section 10.1) with nodes colored red and black in a way that satisfies the following properties:

***Root Property*:** The root is black.

***External Property*:** Every external node is black.

***Internal Property*:** The children of a red node are black.

Depth Property*:** All the external nodes have the same ***black depth, defined as the number of black ancestors minus one. (Recall that a node is an ancestor of itself.)

An example of a red-black tree is shown in Figure 10.27.

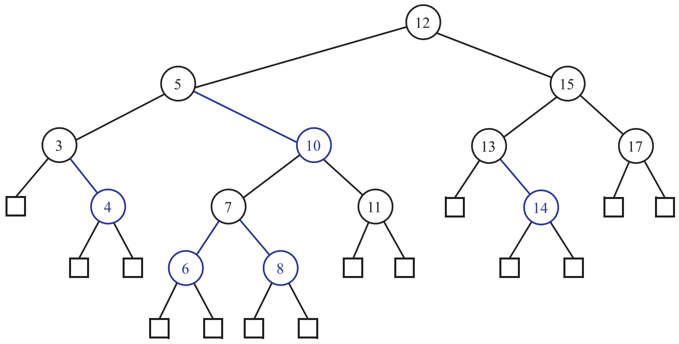


Figure 10.27: Red-black tree associated with the $(2,4)$ tree of Figure 10.21. Each external node of this red-black tree has 4 black ancestors (including itself); hence, it has black depth 3. We use the color blue instead of red. Also, we use the convention of giving an edge of the tree the same color as the child node.

As for previous types of search trees, we assume that entries are stored at the internal nodes of a red-black tree, with the external nodes being empty placeholders. Also, we assume that the external nodes are actual nodes, but we note that, at the expense of slightly more complicated methods, external nodes could be *null*.

We can make the red-black tree definition more intuitive by noting an interesting correspondence between red-black trees and $(2,4)$ trees as illustrated in Figure 10.28. Namely, given a red-black tree, we can construct a corresponding $(2,4)$ tree by merging every red node v into its parent and storing the entry from v at its parent. Conversely, we can transform any $(2,4)$ tree into a corresponding red-black tree by coloring each node black and performing the following transformation for each internal node v:

- If v is a 2-node, then keep the (black) children of v as is
- If v is a 3-node, then create a new red node w, give v's first two (black) children to w, and make w and v's third child be the two children of v
- If v is a 4-node, then create two new red nodes w and z, give v's first two (black) children to w, give v's last two (black) children to z, and make w and z be the two children of v

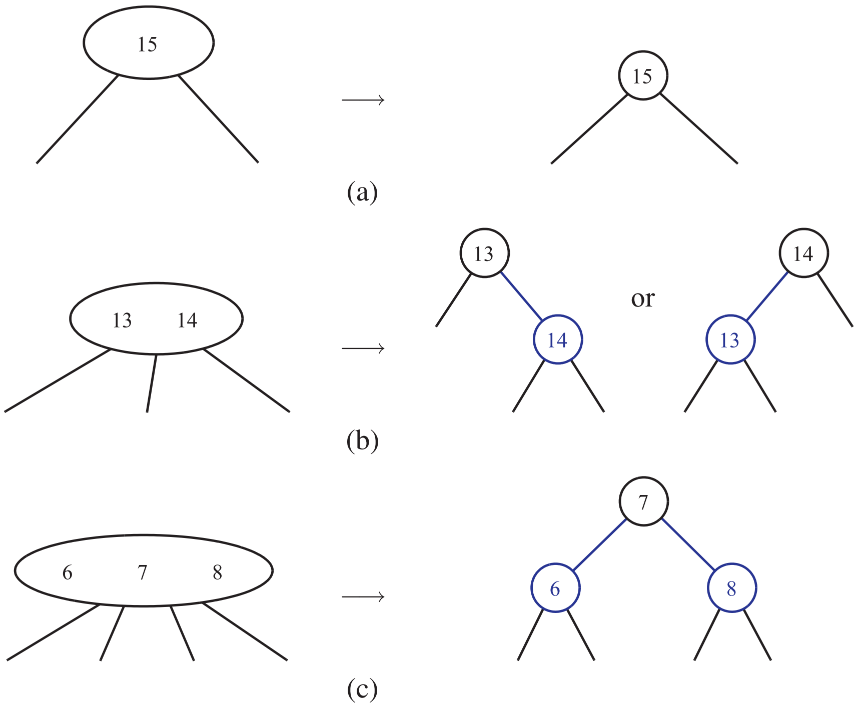


Figure 10.28: Correspondence between a $(2,4)$ tree and a red-black tree: (a) 2-node; (b) 3-node; (c) 4-node.

The correspondence between $(2,4)$ trees and red-black trees provides important intuition that we use in our discussion of how to perform updates in red-black trees. In fact, the update algorithms for red-black trees are mysteriously complex without this intuition.

Proposition 10.9: *The height of a red-black tree storing n entries is $O(\log n)$.*

Justification: Let T be a red-black tree storing n entries, and let h be the height of T. We justify this proposition by establishing the following fact

$$\log(n+1) \le h \le 2\log(n+1).$$

Let d be the common black depth of all the external nodes of T. Let T' be the $(2,4)$ tree associated with T, and let h' be the height of T'. Because of the correspondence between red-black trees and $(2,4)$ trees, we know that $h' = d$. Hence, by Proposition 10.8, $d = h' \le \log(n+1)$. By the internal node property, $h \le 2d$. Thus, we obtain $h \le 2\log(n+1)$. The other inequality, $\log(n+1) \le h$, follows from Proposition 7.10 and the fact that T has n internal nodes. ■

We assume that a red-black tree is realized with a linked structure for binary trees (Section 7.3.4), in which we store a map entry and a color indicator at each node. Thus, the space requirement for storing n keys is $O(n)$. The algorithm for searching in a red-black tree T is the same as that for a standard binary search tree (Section 10.1). Thus, searching in a red-black tree takes $O(\log n)$ time.

10.5.1 Update Operations

Performing the update operations in a red-black tree is similar to that of a binary search tree, except that we must additionally restore the color properties.

Insertion

Now consider the insertion of an entry with key k into a red-black tree T, keeping in mind the correspondence between T and its associated $(2,4)$ tree T' and the insertion algorithm for T'. The algorithm initially proceeds as in a binary search tree (Section 10.1.2). Namely, we search for k in T until we reach an external node of T, and we replace this node with an internal node z, storing (k,x) and having two external-node children. If z is the root of T, we color z black, else we color z red. We also color the children of z black. This action corresponds to inserting (k,x) into a node of the $(2,4)$ tree T' with external children. In addition, this action preserves the root, external, and depth properties of T, but it may violate the internal property. Indeed, if z is not the root of T and the parent v of z is red, then we have a parent and a child (namely, v and z) that are both red. Note that by the root property, v cannot be the root of T, and by the internal property (which was previously satisfied), the parent u of v must be black. Since z and its parent are red, but z's grandparent u is black, we call this violation of the internal property a ***double red*** at node z.

To remedy a double red, we consider two cases.

Case 1: ***The Sibling*** **w** ***of*** **v** ***is Black.*** (See Figure 10.29.) In this case, the double red denotes the fact that we have created in our red-black tree T a malformed replacement for a corresponding 4-node of the $(2,4)$ tree T', which has as its children the four black children of u, v, and z. Our malformed replacement has one red node (v) that is the parent of another red node (z), while we want it to have the two red nodes as siblings instead. To fix this problem, we perform a ***trinode restructuring*** of T. The trinode restructuring is done by the operation restructure(z), which consists of the following steps (see again Figure 10.29; this operation is also discussed in Section 10.2):

- Take node z, its parent v, and grandparent u, and temporarily relabel them as a, b, and c, in left-to-right order, so that a, b, and c will be visited in this order by an inorder tree traversal.
- Replace the grandparent u with the node labeled b, and make nodes a and c the children of b, keeping inorder relationships unchanged.

After performing the restructure(z) operation, we color b black and we color a and c red. Thus, the restructuring eliminates the double red problem.

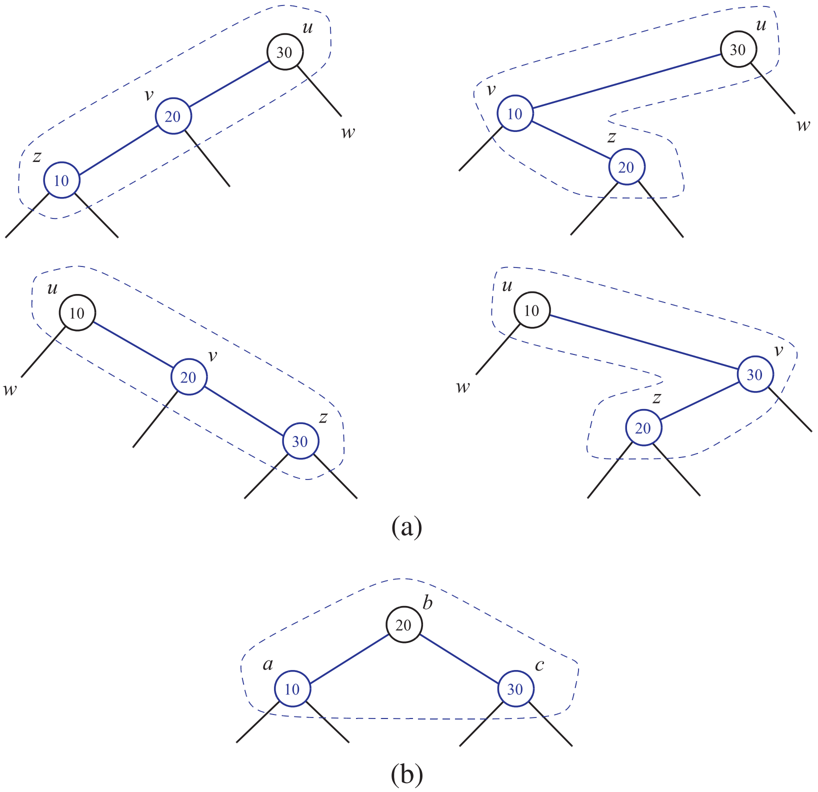


Figure 10.29: Restructuring a red-black tree to remedy a double red: (a) the four configurations for u, v, and z before restructuring; (b) after restructuring.

Case 2: ***The Sibling* w *of* v *is Red.*** (See Figure 10.30.) In this case, the double red denotes an overflow in the corresponding $(2,4)$ tree T. To fix the problem, we perform the equivalent of a split operation. Namely, we do a ***recoloring***: we color v and w black and their parent u red (unless u is the root, in which case, it is colored black). It is possible that, after such a recoloring, the double red problem reappears, although higher up in the tree T, since u may have a red parent. If the double red problem reappears at u, then we repeat the consideration of the two cases at u. Thus, a recoloring either eliminates the double red problem at node z, or propagates it to the grandparent u of z. We continue going up T performing recolorings until we finally resolve the double red problem (with either a final recoloring or a trinode restructuring). Thus, the number of recolorings caused by an insertion is no more than half the height of tree T, that is, no more than $\log(n+1)$ by Proposition 10.9.

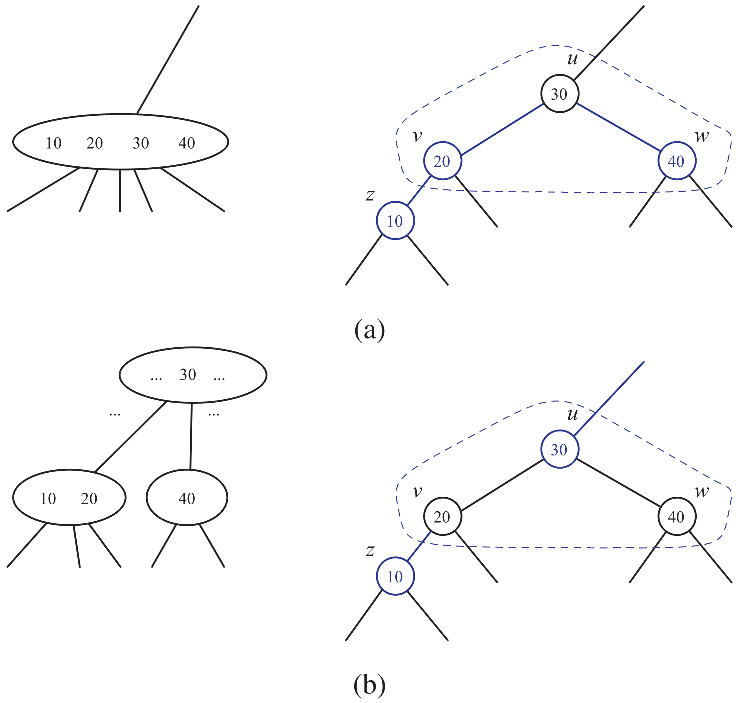


Figure 10.30: Recoloring to remedy the double red problem: (a) before recoloring and the corresponding 5-node in the associated $(2,4)$ tree before the split; (b) after the recoloring (and corresponding nodes in the associated $(2,4)$ tree after the split).

Figures 10.31 and 10.32 show a sequence of insertion operations in a red-black tree.

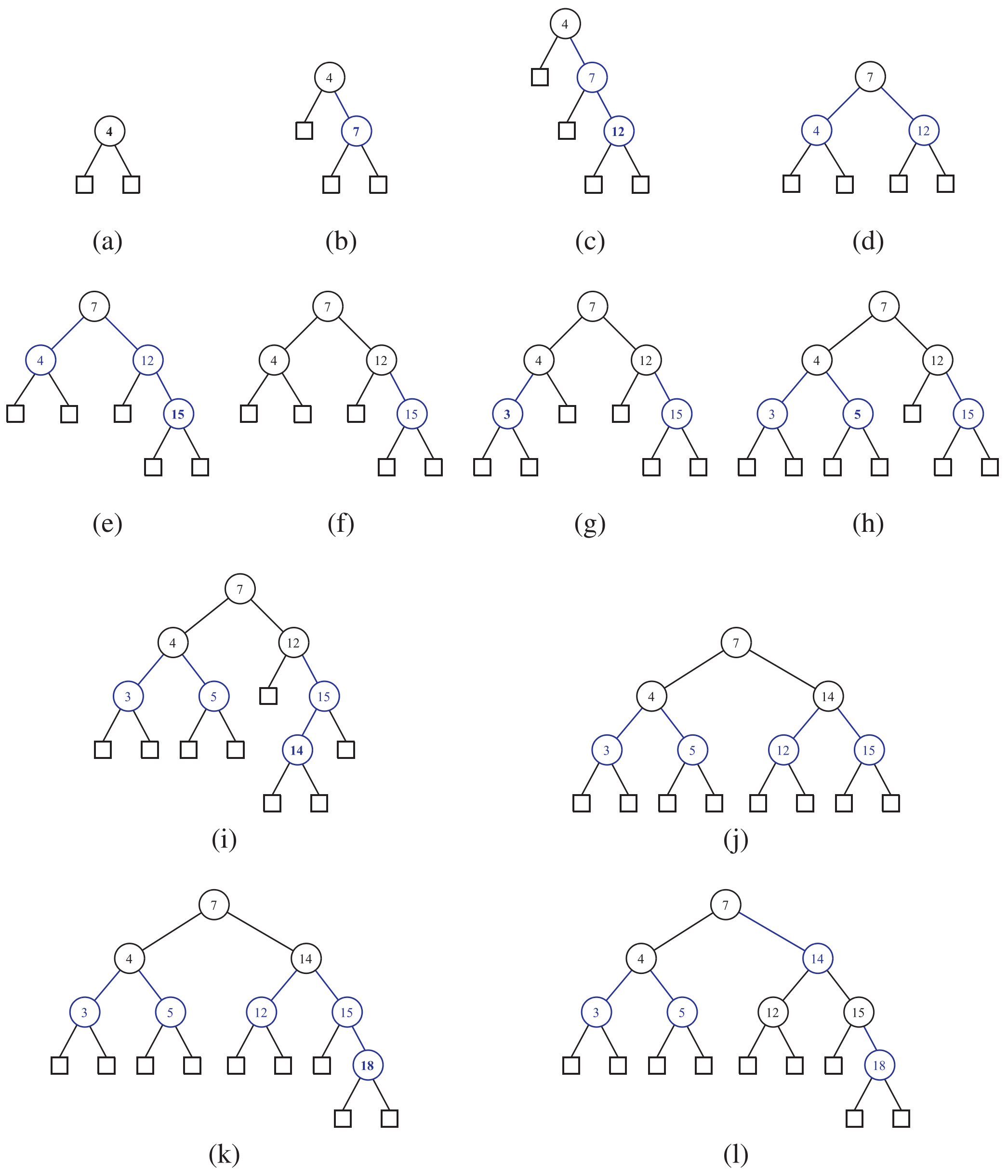


Figure 10.31: A sequence of insertions in a red-black tree: (a) initial tree; (b) insertion of 7; (c) insertion of 12, which causes a double red; (d) after restructuring; (e) insertion of 15, which causes a double red; (f) after recoloring (the root remains black); (g) insertion of 3; (h) insertion of 5; (i) insertion of 14, which causes a double red; (j) after restructuring; (k) insertion of 18, which causes a double red; (l) after recoloring. (Continues in Figure 10.32.)

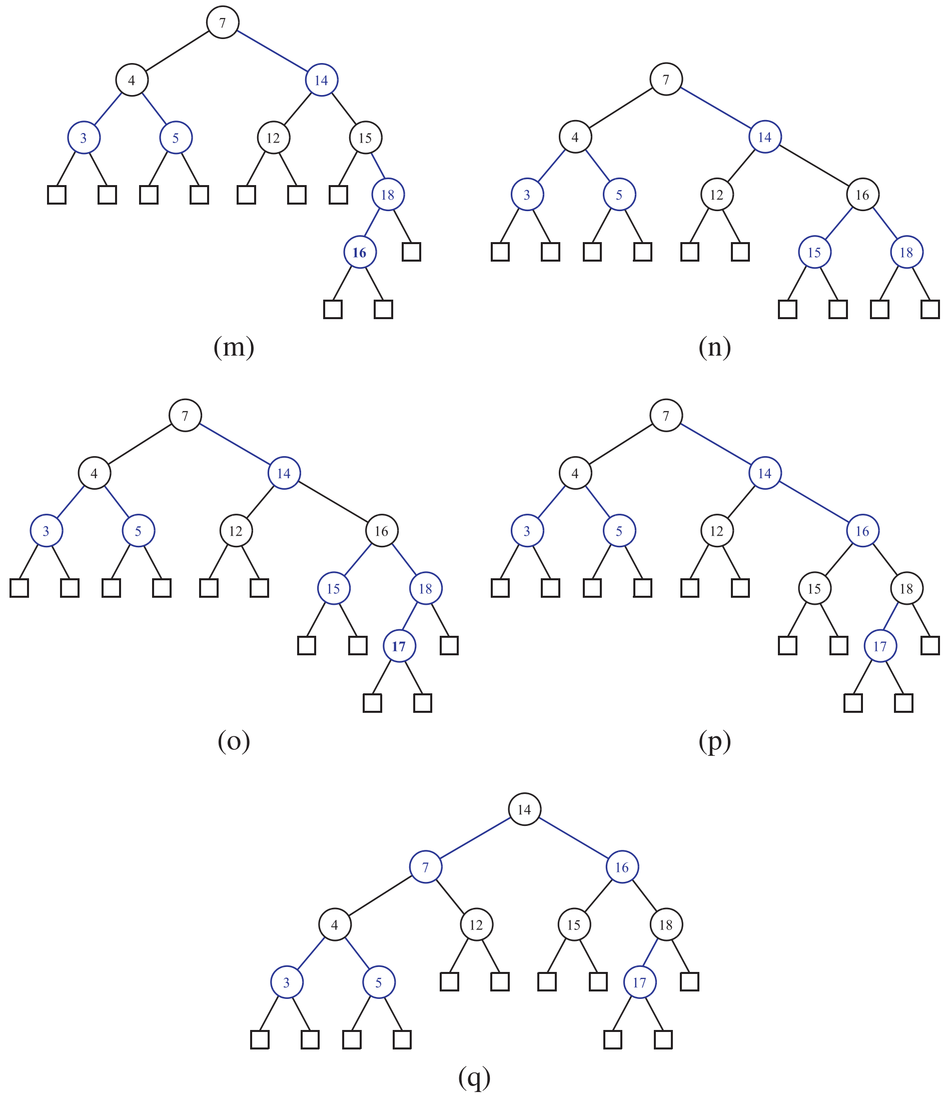


Figure 10.32: A sequence of insertions in a red-black tree: (m) insertion of 16, which causes a double red; (n) after restructuring; (o) insertion of 17, which causes a double red; (p) after recoloring there is again a double red, to be handled by a restructuring; (q) after restructuring. (Continued from Figure 10.31.)

The cases for insertion imply an interesting property for red-black trees. Namely, since the Case 1 action eliminates the double-red problem with a single trinode restructuring and the Case 2 action performs no restructuring operations, at most one restructuring is needed in a red-black tree insertion. By the above analysis and the fact that a restructuring or recoloring takes $O(1)$ time, we have the following.

Proposition 10.10: *The insertion of a key-value entry in a red-black tree storing n entries can be done in $O(\log n)$ time and requires $O(\log n)$ recolorings and one trinode restructuring (a* restructure *operation).*

Removal

Suppose now that we are asked to remove an entry with key k from a red-black tree T. Removing such an entry initially proceeds like a binary search tree (Section 10.1.2). First, we search for a node u storing such an entry. If node u does not have an external child, we find the internal node v following u in the inorder traversal of T, move the entry at v to u, and perform the removal at v. Thus, we may consider only the removal of an entry with key k stored at a node v with an external child w. Also, as we did for insertions, we keep in mind the correspondence between red-black tree T and its associated $(2,4)$ tree T' (and the removal algorithm for T').

To remove the entry with key k from a node v of T with an external child w we proceed as follows. Let r be the sibling of w and x be the parent of v. We remove nodes v and w, and make r a child of x. If v was red (hence r is black) or r is red (hence v was black), we color r black and we are done. If, instead, r is black and v was black, then, to preserve the depth property, we give r a fictitious ***double black*** color. We now have a color violation, called the double black problem. A double black in T denotes an underflow in the corresponding $(2,4)$ tree T'. Recall that x is the parent of the double black node r. To remedy the double-black problem at r, we consider three cases.

Case 1: ***The Sibling* y *of* r *is Black and Has a Red Child* z.** (See Figure 10.33.) Resolving this case corresponds to a transfer operation in the $(2,4)$ tree T'. We perform a ***trinode restructuring*** by means of operation restructure(z). Recall that the operation restructure(z) takes the node z, its parent y, and grandparent x, labels them temporarily left to right as a, b, and c, and replaces x with the node labeled b, making it the parent of the other two. (See the description of restructure in Section 10.2.) We color a and c black, give b the former color of x, and color r black. This trinode restructuring eliminates the double black problem. Hence, at most one restructuring is performed in a removal operation in this case.

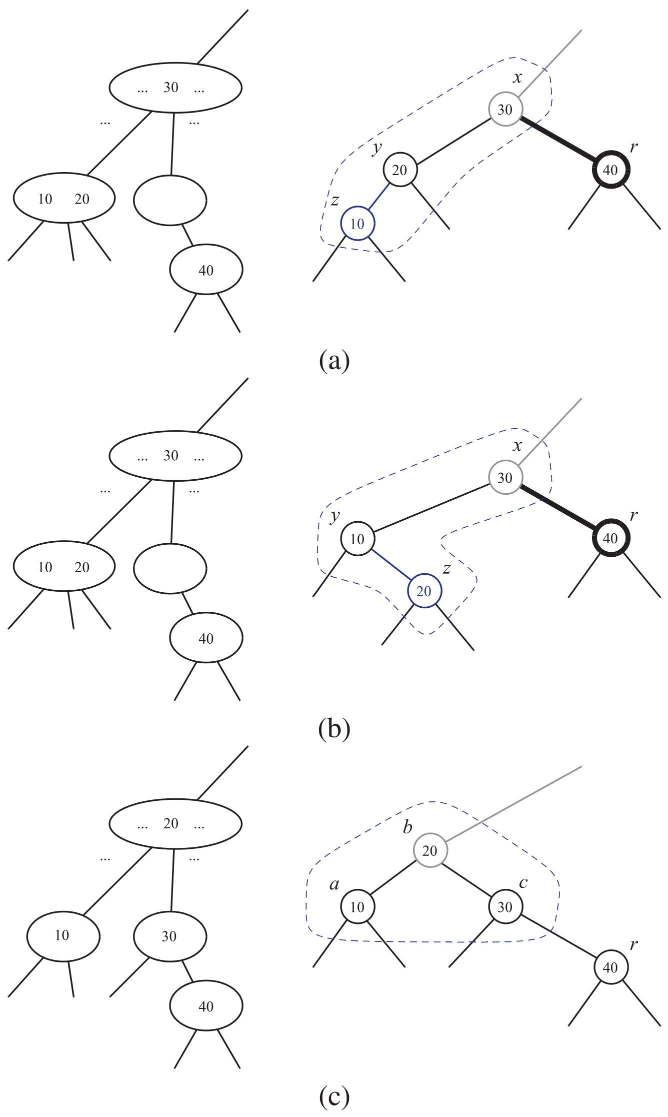
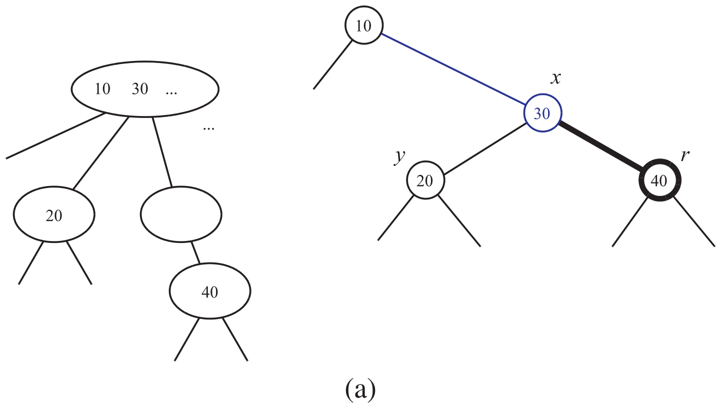


Figure 10.33: Restructuring of a red-black tree to remedy the double black problem: (a) and (b) configurations before the restructuring, where r is a right child and the associated nodes in the corresponding $(2,4)$ tree before the transfer (two other symmetric configurations where r is a left child are possible); (c) configuration after the restructuring and the associated nodes in the corresponding $(2,4)$ tree after the transfer. The grey color for node x in parts (a) and (b) and for node b in part (c) denotes the fact that this node may be colored either red or black.

Case 2: ***The Sibling*** **y** ***of*** **r** ***is Black and Both Children of*** **y** ***Are Black.*** (See Figures 10.34 and 10.35.) Resolving this case corresponds to a fusion operation in the corresponding $(2,4)$ tree T'. We do a ***recoloring***; we color r black, we color y red, and, if x is red, we color it black (Figure 10.34); otherwise, we color x ***double black*** (Figure 10.35). Hence, after this recoloring, the double black problem may reappear at the parent x of r. (See Figure 10.35.) That is, this recoloring either eliminates the double black problem or propagates it into the parent of the current node. We then repeat a consideration of these three cases at the parent. Thus, since Case 1 performs a trinode restructuring operation and stops (and, as we will soon see, Case 3 is similar), the number of recolorings caused by a removal is no more than $\log(n+1)$.



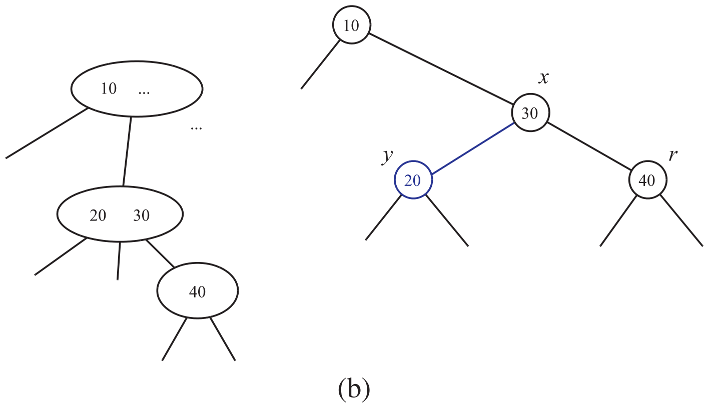
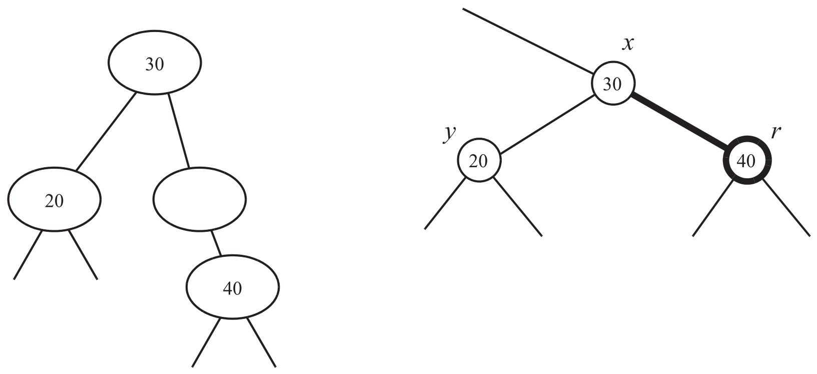
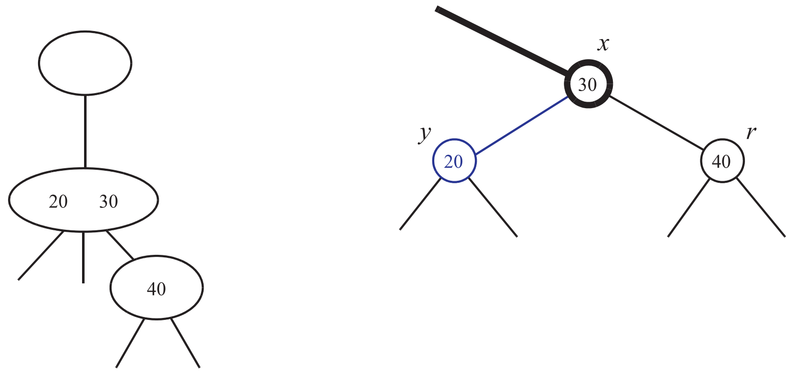


Figure 10.34: Recoloring of a red-black tree that fixes the double black problem: (a) before the recoloring and corresponding nodes in the associated $(2,4)$ tree before the fusion (other similar configurations are possible); (b) after the recoloring and corresponding nodes in the associated $(2,4)$ tree after the fusion.



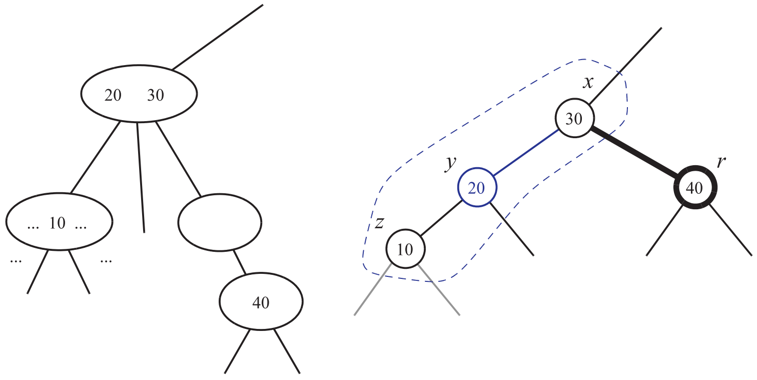
(a)



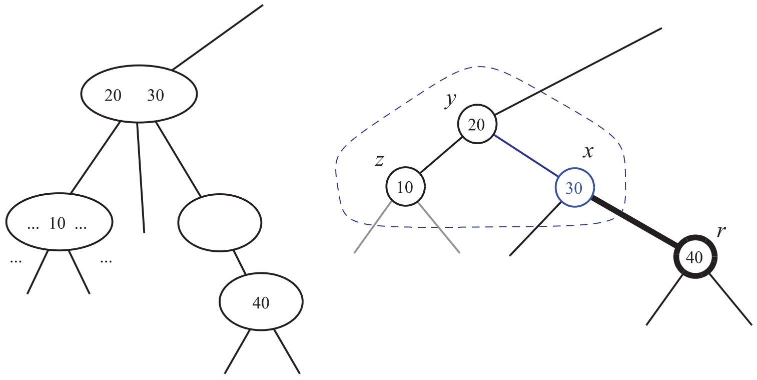
(b)

Figure 10.35: Recoloring of a red-black tree that propagates the double black problem: (a) configuration before the recoloring and corresponding nodes in the associated $(2,4)$ tree before the fusion (other similar configurations are possible); (b) configuration after the recoloring and corresponding nodes in the associated $(2,4)$ tree after the fusion.

Case 3: ***The Sibling* y *of* r *Is Red.*** (See Figure 10.36.) In this case, we perform an ***adjustment*** operation, as follows. If y is the right child of x, let z be the right child of y; otherwise, let z be the left child of y. Execute the trinode restructuring operation restructure(z), which makes y the parent of x. Color y black and x red. An adjustment corresponds to choosing a different representation of a 3-node in the $(2,4)$ tree T'. After the adjustment operation, the sibling of r is black, and either Case 1 or Case 2 applies, with a different meaning of x and y. Note that if Case 2 applies, the double-black problem cannot reappear. Thus, to complete Case 3 we make one more application of either Case 1 or Case 2 above and we are done. Therefore, at most one adjustment is performed in a removal operation.



(a)



(b)

Figure 10.36: Adjustment of a red-black tree in the presence of a double black problem: (a) configuration before the adjustment and corresponding nodes in the associated $(2,4)$ tree (a symmetric configuration is possible); (b) configuration after the adjustment with the same corresponding nodes in the associated $(2,4)$ tree.

From the above algorithm description, we see that the tree updating needed after a removal involves an upward march in the tree T, while performing at most a constant amount of work (in a restructuring, recoloring, or adjustment) per node. Thus, since any changes we make at a node in T during this upward march takes $O(1)$ time (because it affects a constant number of nodes), we have the following.

Proposition 10.11: *The algorithm for removing an entry from a red-black tree with n entries takes $O(\log n)$ time and performs $O(\log n)$ recolorings and at most one adjustment plus one additional trinode restructuring. Thus, it performs at most two* restructure *operations.*

In Figures 10.37 and 10.38, we show a sequence of removal operations on a red-black tree. We illustrate Case 1 restructurings in Figure 10.37(c) and (d). We illustrate Case 2 recolorings at several places in Figures 10.37 and 10.38. Finally, in Figure 10.38(i) and (j), we show an example of a Case 3 adjustment.

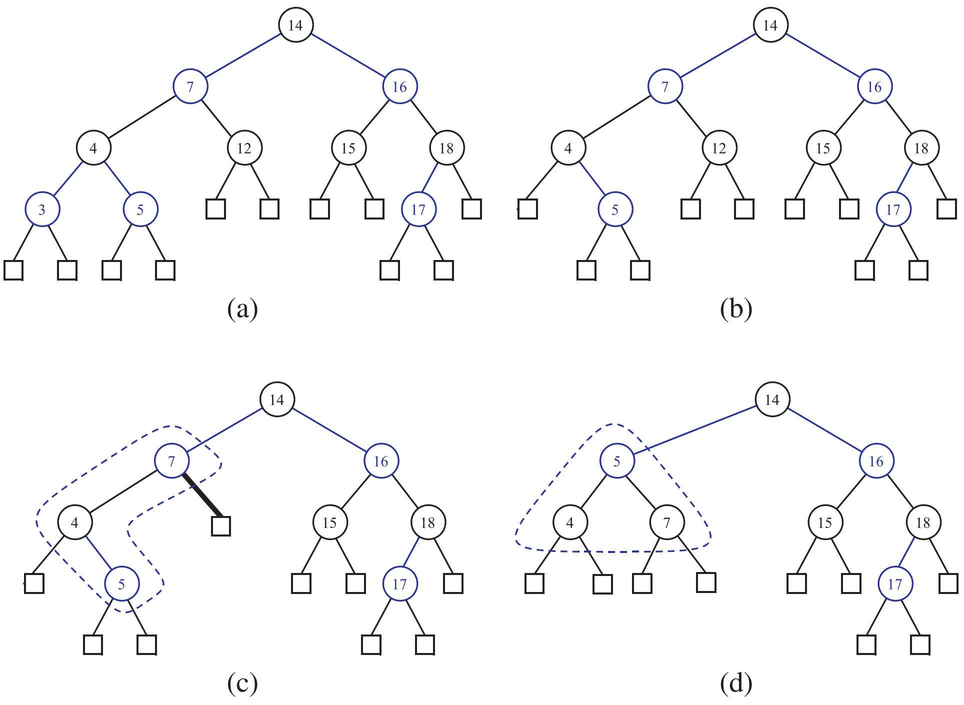


Figure 10.37: Sequence of removals from a red-black tree: (a) initial tree; (b) removal of 3; (c) removal of 12, causing a double black (handled by restructuring); (d) after restructuring. (Continues in Figure 10.38.)

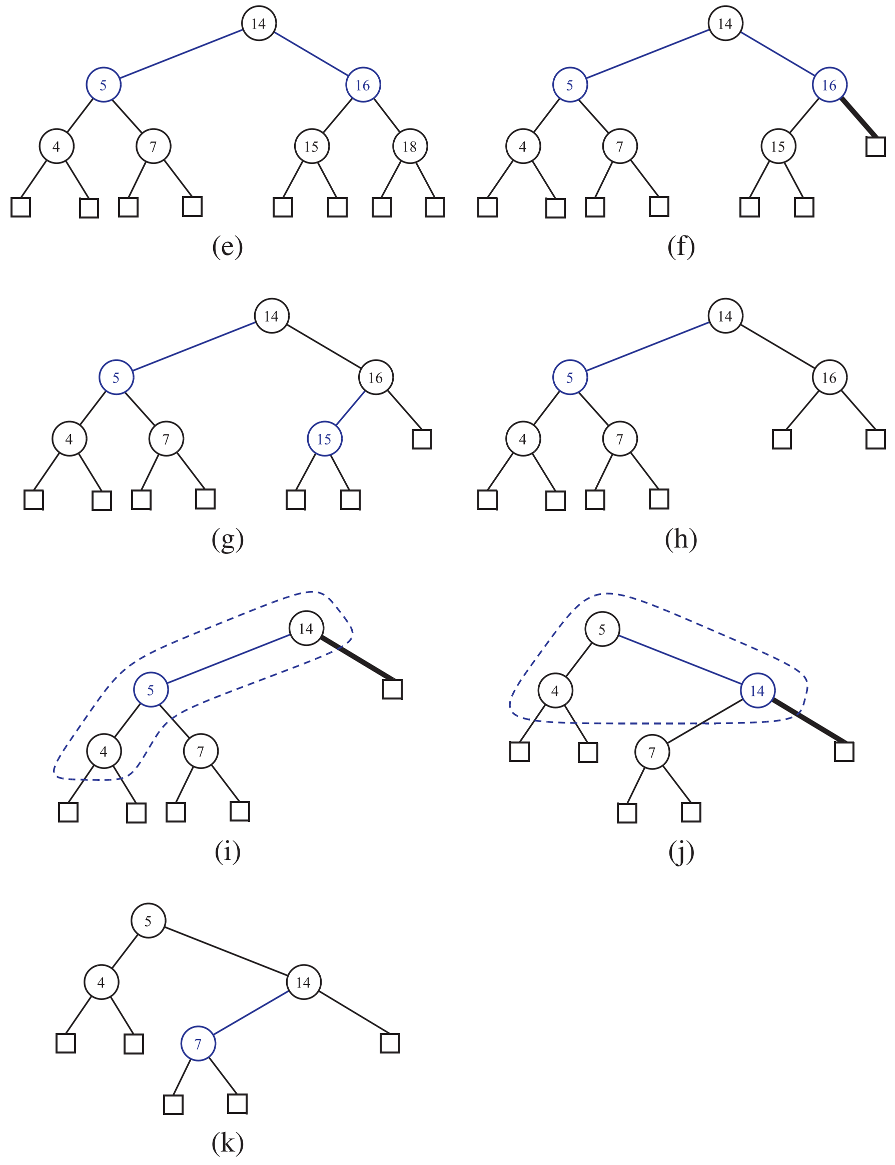


Figure 10.38: Sequence of removals in a red-black tree : (e) removal of 17; (f) removal of 18, causing a double black (handled by recoloring); (g) after recoloring; (h) removal of 15; (i) removal of 16, causing a double black (handled by an adjustment); (j) after the adjustment the double black needs to be handled by a recoloring; (k) after the recoloring. (Continued from Figure 10.37.)

Performance of Red-Black Trees

Table 10.4 summarizes the running times of the main operations of a map realized by means of a red-black tree. We illustrate the justification for these bounds in Figure 10.39.

Operation	*Time*
size, empty	$O(1)$
find, insert, erase	$O(\log n)$

Table 10.4: Performance of an n-entry map realized by a red-black tree. The space usage is $O(n)$.

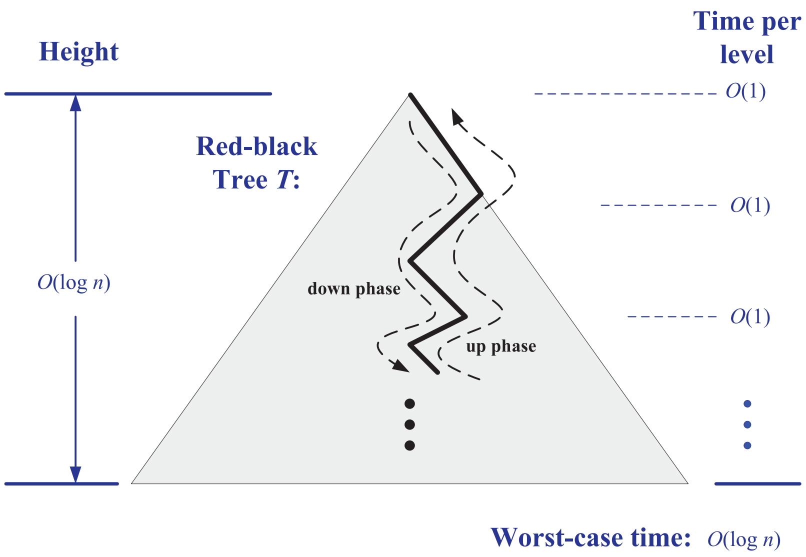


Figure 10.39: The running time of searches and updates in a red-black tree. The time performance is $O(1)$ per level, broken into a down phase, which typically involves searching, and an up phase, which typically involves recolorings and performing local trinode restructurings (rotations).

Thus, a red-black tree achieves logarithmic worst-case running times for both searching and updating in a map. The red-black tree data structure is slightly more complicated than its corresponding $(2,4)$ tree. Even so, a red-black tree has a conceptual advantage that only a constant number of trinode restructurings are ever needed to restore the balance in a red-black tree after an update.

10.5.2 C++ Implementation of a Red-Black Tree

In this section, we discuss a C++ implementation of the dictionary ADT by means of a red-black tree. It is interesting to note that the C++ Standard Template Library uses a red-black tree in its implementation of its classes map and multimap. The difference between the two is similar to the difference between our map and dictionary ADTs. The STL map class does not allow entries with duplicate keys, whereas the STL multimap does. There is a significant difference, however, in the behavior of the map's insert(k,x) function and our map's put(k,x) function. If the key k is not present, both functions insert the new entry (k,x) in the map. If the key is already present, the STL map simply ignores the request, and the current entry is unchanged. In contrast, our put function replaces the existing value with the new value x. The implementation presented in this section allows for multiple keys.

We present the major portions of the implementation in this section. To keep the presentation concise, we have omitted the implementations of a number of simpler utility functions.

We begin by presenting the enhanced entry class, called RBEntry. It is derived from the entry class of Code Fragment 10.3. It inherits the key and value members, and it defines a member variable *col*, which stores the color of the node. The color is either RED or BLACK. It provides member functions for accessing and setting this value. These functions have been protected, so a user cannot access them, but RBTree can.

```
enum Color {RED, BLACK};                              // node colors

template <typename E>
class RBEntry : public E {                            // a red-black entry
private:
  Color col;                                          // node color
protected:                                            // local types
  typedef typename E::Key K;                          // key type
  typedef typename E::Value V;                        // value type
  Color color() const { return col; }                 // get color
  bool isRed() const { return col == RED; }
  bool isBlack() const { return col == BLACK; }
  void setColor(Color c) { col = c; }
public:                                               // public functions
  RBEntry(const K& k = K(), const V& v = V())  // constructor
    : E(k,v), col(BLACK) { }
  friend class RBTree<E>;                             // allow RBTree access
};
```

Code Fragment 10.19: A key-value entry for class RBTree, containing the associated node's color.

In Code Fragment 10.20, we present the class definition for RBTree. The declaration is almost entirely analogous to that of AVLTree, except that the utility functions used to maintain the structure are different. We have chosen to present only the two most interesting utility functions, remedyDoubleRed and remedyDoubleBlack. The meanings of most of the omitted utilities are easy to infer. (For example hasTwoExternalChildren(v) determines whether a node v has two external children.)

```
template <typename E>                                  // a red-black tree
class RBTree : public SearchTree< RBEntry<E> > {
public:                                                // public types
  typedef RBEntry<E> RBEntry;                          // an entry
  typedef typename SearchTree<RBEntry>::Iterator Iterator; // an iterator
protected:                                             // local types
  typedef typename RBEntry::Key K;                     // a key
  typedef typename RBEntry::Value V;                   // a value
  typedef SearchTree<RBEntry> ST;                      // a search tree
  typedef typename ST::TPos TPos;                      // a tree position
public:                                                // public functions
  RBTree();                                            // constructor
  Iterator insert(const K& k, const V& x);             // insert (k,x)
  void erase(const K& k) throw(NonexistentElement); // remove key k entry
  void erase(const Iterator& p);                       // remove entry at p
protected:                                             // utility functions
  void remedyDoubleRed(const TPos& z);                 // fix double-red z
  void remedyDoubleBlack(const TPos& r);               // fix double-black r
  // ...(other utilities omitted)
};
```

Code Fragment 10.20: Class RBTree, which implements a dictionary ADT using a red-black tree.

We first discuss the implementation of the function insert(k,x), which is given in Code Fragment 10.21. We invoke the inserter utility function of SearchTree, which returns the position of the inserted node. If this node is the root of the search tree, we set its color to black. Otherwise, we set its color to red and check whether restructuring is needed by invoking remedyDoubleRed.

This latter utility performs the necessary checks and restructuring presented in the discussion of insertion in Section 10.5.1. Let z denote the location of the newly inserted node. If both z and its parent are red, we need to remedy the situation. To do so, we consider two cases. Let v denote z's parent and let w be v's sibling. If w is black, we fall under Case 1 of the insertion update procedure. We apply restructuring at z. The top vertex of the resulting subtree, denoted by v, is set to black, and its two children are set to red.

On the other hand, if w is red, then we fall under Case 2 of the update procedure.

We resolve the situation by coloring both v and its sibling w black. If their common parent is not the root, we set its color to red. This may induce another double-red problem at v's parent u, so we invoke the function recursively on u.

```
/* RBTree⟨E⟩ :: */                                  // insert (k,x)
  Iterator insert(const K& k, const V& x) {
    TPos v = inserter(k, x);                         // insert in base tree
    if (v == ST::root())
      setBlack(v);                                   // root is always black
    else {
      setRed(v);
      remedyDoubleRed(v);                            // rebalance if needed
    }
    return Iterator(v);
  }

/* RBTree⟨E⟩ :: */                                  // fix double-red z
  void remedyDoubleRed(const TPos& z) {
    TPos v = z.parent();                             // v is z's parent
    if (v == ST::root() || v->isBlack()) return;     // v is black, all ok
                                                     // z, v are double-red
    if (sibling(v)->isBlack()) {                     // Case 1: restructuring
      v = restructure(z);
      setBlack(v);                                   // top vertex now black
      setRed(v.left()); setRed(v.right());           // set children red
    }
    else {                                           // Case 2: recoloring
      setBlack(v); setBlack(sibling(v));             // set v and sibling black
      TPos u = v.parent();                           // u is v's parent
      if (u == ST::root()) return;
      setRed(u);                                     // make u red
      remedyDoubleRed(u);                            // may need to fix u now
    }
  }
```

Code Fragment 10.21: The functions related to insertion for class RBTree. The function insert invokes the inserter utility function, which was given in Code Fragment 10.10.

Finally, in Code Fragment 10.22, we present the implementation of the removal function for the red-black tree. (We have omitted the simpler iterator-based erase function.) The removal follows the process discussed in Section 10.5.1. We first search for the key to be removed, and generate an exception if it is not found. Otherwise, we invoke the eraser utility of class SearchTree, which returns the position of the node r that replaced the deleted node. If either r or its former parent was red, we color r black and we are done. Otherwise, we face a potential double-black problem. We handle this by invoking the function remedyDoubleBlack.

```
/* RBTree⟨E⟩ :: */                                     // remove key k entry
  void erase(const K& k) throw(NonexistentElement) {
    TPos u = finder(k, ST::root());                    // find the node
    if (Iterator(u) == ST::end())
      throw NonexistentElement("Erase of nonexistent");
    TPos r = eraser(u);                                // remove u
    if (r == ST::root() || r−>isRed() || wasParentRed(r))
      setBlack(r);                                     // fix by color change
    else                                               // r, parent both black
      remedyDoubleBlack(r);                            // fix double-black r
  }

/* RBTree⟨E⟩ :: */                                     // fix double-black r
  void remedyDoubleBlack(const TPos& r) {
    TPos x = r.parent();                               // r's parent
    TPos y = sibling(r);                               // r's sibling
    if (y−>isBlack()) {
      if (y.left()−>isRed() || y.right()−>isRed()) {   // Case 1: restructuring
                                                       // z is y's red child
        TPos z = (y.left()−>isRed() ? y.left() : y.right());
        Color topColor = x−>color();                   // save top vertex color
        z = restructure(z);                            // restructure x,y,z
        setColor(z, topColor);                         // give z saved color
        setBlack(r);                                   // set r black
        setBlack(z.left()); setBlack(z.right());       // set z's children black
      }
      else {                                           // Case 2: recoloring
        setBlack(r); setRed(y);                        // r=black, y=red
        if (x−>isBlack() && !(x == ST::root()))
        remedyDoubleBlack(x);                          // fix double-black x
        setBlack(x);
      }
    }
    else {                                             // Case 3: adjustment
      TPos z = (y == x.right() ? y.right() : y.left()); // grandchild on y's side
      restructure(z);                                  // restructure x,y,z
      setBlack(y); setRed(x);                          // y=black, x=red
      remedyDoubleBlack(r);                            // fix r by Case 1 or 2
    }
  }
```

Code Fragment 10.22: The functions related to removal for class RBTree. The function erase invokes the eraser utility function, which was given in Code Fragment 10.11.